

GhostDB: Hiding Data from Prying Eyes

Christophe Salperwyck^{*}, Nicolas Anciaux^{*}, Mehdi Benzine^{*,**},
Luc Bouganim^{*}, Philippe Pucheral^{*,**}, Dennis Shasha^{*,***}

^{*} INRIA Rocquencourt

Le Chesnay, France

<Fname.Lname>@inria.fr

^{**} PRiSM Laboratory

University of Versailles, France

<Fname.Lname>@prism.uvsq.fr

^{***} Courant Institute of Mathematical Sciences

New York University, New York, USA

shasha@cs.nyu.edu

ABSTRACT

Imagine that you have been entrusted with private data, such as corporate product information, sensitive government information, or symptom and treatment information about hospital patients. You may want to issue queries whose result will combine private and public data, but private data must not be revealed, say, to the prying eyes of some insurance fraudster. GhostDB is an architecture and system to achieve this.

You carry private data in a smart USB device (a large Flash persistent store combined with a tamper and snoop-resistant CPU and small RAM). When the key is plugged in, you can issue queries that link private and public data and be sure that the only information revealed to a potential spy is which queries you pose and the public data you access. Queries linking public and private data entail novel distributed processing techniques on extremely unequal devices (standard computer and smart USB device) in which data flows in only one direction: from public to private. This demonstration shows GhostDB's query processing in action.

1. MOTIVATION

Traditional security procedures do not offer sufficient protection against data theft [4]. Recent academic work does provide additional guarantees under specific assumptions regarding where the trust resides in the system. Hippocratic databases ensure that personal data are used in compliance with the purpose for which the donor gave his consent [2] but require the database server to be trusted. Encrypted databases require either trusting the server [7] or the clients [6] depending on the place decryption occurs. Entire databases can also be hosted in secure hardware [8][10] but this solution applies only to small mono-user databases. Finally, an alternative solution can be anonymizing the data [9] at the price of lesser data accuracy and usability.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Database Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.

VLDB '07, September 23-28, 2007, Vienna, Austria.

Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

We propose a different approach to protecting sensitive data, recognizing that truly private data is a small portion of all the data that may be of interest in an application. The basic idea is to remove all sensitive data from internet-accessible places and allocate that data to trusted devices with strong guarantees against spying, much stronger than a spyware-prone laptop. We do this with minimal changes to schema definitions and no changes to the SQL query text.

2. HIDDEN vs. VISIBLE

In the specific demo scenario, we imagine that a user Bob has been entrusted with sensitive health data about diabetes patients in a hospital. The security administrator has declared certain sensitive columns to be “hidden” (should never be revealed to the public) and other columns to be “visible” (may be public, or at least visible to general hospital personnel). This is done through standard create table commands except that some field definitions have the extra keyword “hidden”. For example:

```
CREATE TABLE Visit (  
  VisID INTEGER PRIMARY KEY,  
  Date DATE,  
  Purpose CHAR(100) HIDDEN,  
  DocID REFERENCES Doctor(DocID) HIDDEN,  
  PatID REFERENCES Patient(PatID) HIDDEN);
```

Primary keys as well as visible fields can be stored at any place, like a public server or a personal computer and are therefore vulnerable to snooping (see left part of Figure 1). The hidden fields are hosted by Bob's USB device (in the circle in figure 1) and then protected from any form of spying. The primary keys of all tables are replicated in the USB device to allow for queries combining visible and hidden data. In the demonstration scenario, we make foreign keys hidden because they offer the possibility of linking sensitive records, but different choices may make sense in other contexts. The USB device is assumed to be initially loaded in a secure setting.

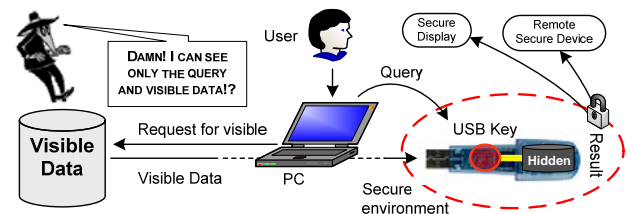


Figure 1. GhostDB mode of operation.

GhostDB works as follows. Queries are issued on the personal computer and visible data is found on the PC and/or on remote server(s). Visible data enters the USB device but all executions involving hidden data or the combination of hidden and visible data occur on the USB device. Neither hidden data nor intermediate results ever leave that device.

In eventual deployment Bob needs a secure rendering platform. This could be the USB device itself (some smart memory sticks already hold a small LCD screen), possibly improved by technologies such as fiber carbon [5]. This could also be an external palm-style screen or tablet connected to the key or even the screen of the terminal the key is plugged into if a secure channel can be established with the video card (Digital Right Management companies are investigating this solution). Another mode of operation is sending the result to a remote secure application through a secure socket connection. Whichever the choice, the net effect is that Bob reveals to a potential spy only the queries he poses and the visible data he accesses.

Queries linking public and private data entail novel distributed processing techniques on extremely unequal devices (standard computer and smart USB device). Beyond experiencing the benefit of the GhostDB approach in terms of data privacy, the visitor to this demonstration will see these processing techniques in action and assess their performance. In the sequel, we first recall from [1] the hardware constraints introduced by the USB device and the indexing and query processing techniques proposed to tackle them. Then we detail the demonstration scenario.

3. SMART USB DEVICE CHALLENGE

GhostDB acquires its tamper resistance from a smart USB device. As pictured in Figure 2, a smart USB device is a combination of a secure chip with a large external Flash memory (Gigabyte sized). The secure chip is usually equipped with a 32 bit RISC processor, memory modules composed of ROM, static RAM (tens of KB), a small quantity of internal stable storage and security modules. Security factors imply that the RAM must be small – the smaller the silicon die, the most difficult it is to snoop or tamper with processing. The Flash memory itself exhibits asymmetric costs for reads and writes. Writes are between 3 to 10 times slower than reads depending on the portion of the page to be read (full page vs. single word) and writes in place are precluded. Finally, the USB2.0 full speed communication throughput reaches 12 Mb/s. High speed (up to 480 Mb/s) is envisioned for future platforms to cope with applications like on-the-fly video decryption.

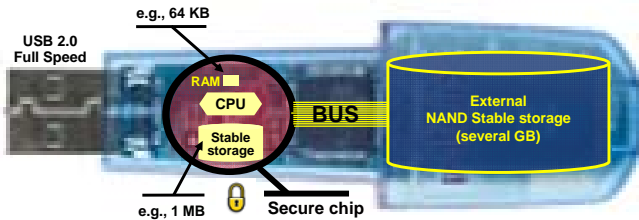


Figure 2. Smart USB device.

These hardware constraints lead (1) to delegate as much work as possible to the PC and the server as long as this processing does not compromise hidden data, and (2) to design query processing techniques accommodating the tiny RAM and the Flash write/read cost ratio.

4. INDEXING AND QUERY PROCESSING

The smart USB device challenge introduced above transforms the privacy preservation problem into a severe performance problem. The problem is twofold: (1) how to compute regular SQL queries (concentrating here on SPJ queries) over arbitrarily large tables under such hardware constraints and (2) how to mix visible and hidden computations efficiently.

In [1], we show that the first part of the problem leads to unacceptable performance with last resort join algorithms (like hash joins) as well as with known indexing techniques like join indices. We then proposed a new indexing model inspired by data warehouse techniques [11]. We illustrate this indexing model on the database schema presented in Figure 3. This schema, which will be used for the demo, is a tree schema in which each patient visit is managed by a specific doctor and prescriptions of medicine are given at each visit. The arrows of Figure 3 represent foreign key relationships and attributes having a superscript H in the schema are hidden.

We propose a set of generalized join indexes known as “Subtree Key Tables” or SKT. There are two SKTs in the Figure: one rooted at Prescription and one at Visit. Each SKT joins all tables in the subtree to the subtree root with the IDs sorted based on the order of IDs in the root table. For example, the SKT rooted at Prescription has PreID, MedID, VisID, DocID, PatID and is sorted based on PreID. This enables a query to directly associate a prescription with the patient to whom it was issued, for example.

To speed up selections, we propose an additional index that we call a “climbing index”. A climbing index on a lower table T in the tree-based database schema maps values to lists of identifiers from T as well as lists of identifiers for each table T’ that is an ancestor of T in the tree. For example, the entry for

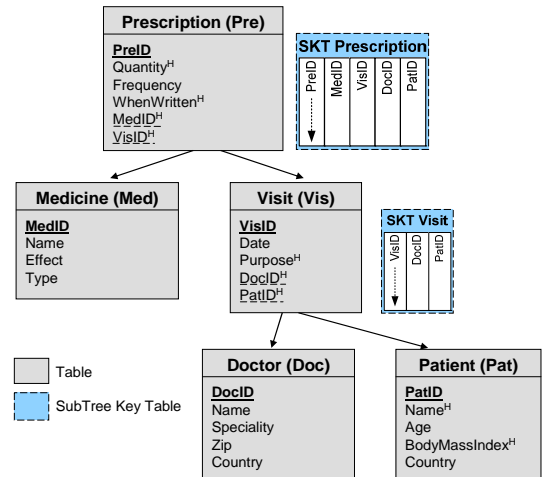


Figure 3. Database schema and SubTree Key Tables.

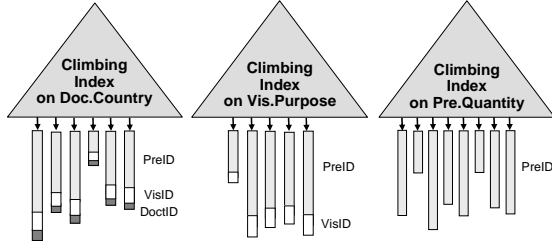


Figure 4. Climbing indexes.

“Spain” in the Doctor.Country index is associated with a list of Doctor identifiers, as usual, and also a list of Visit identifiers and a list of Prescription identifiers to precompute the joins with all tables in the path from Doctor to the root table Prescription (Figure 4). Combined together, SKTs and climbing indexes allow selecting tuples in any table, reaching any other table in the path from this table to the root table in a single step and projecting attributes from any other table of the tree. This benefit in terms of performance and RAM usage comes at an extra cost in terms of Flash storage.

Let us now illustrate how SKTs and climbing indexes are used to execute queries mixing visible and hidden data efficiently, like:

```
SELECT
  Med.Name, Pre.Quantity, Vis.Date
FROM Medicine Med, Prescription Pre, Visit Vis
WHERE
  Vis.Date > 05-11-2006      /*VISIBLE*/
  AND Vis.Purpose = "Sclerosis" /*HIDDEN*/
  AND Med.Type = "Antibiotic" /*VISIBLE*/
  AND Med.MedID = Pre.MedID
  AND Vis.VisID = Pre.VisID;
```

The most intuitive Query Execution Plan (QEP) is pushing selections before joins, as usual, and performing all joins by index. This Pre-filtering strategy consists in: (1) using the climbing index on Vis.Purpose to deliver the list of PreID associated to the value ‘Sclerosis’, (2) delegating the selections on Vis.Date and Med.Type, receiving the two resulting lists of VisID and MedID from outside and transforming these lists into lists of PredID thanks to the climbing index on Vis.VisID and Med.MedID, (3) merging all these PreID lists and finally (4) accessing the SKT_Prescription to get the resulting tuples. For the sake of conciseness, we refer the reader to [1] for an explanation of the final project operation.

If, however, the selectivity of a visible selection is low, traversing the climbing indexes may be a poor choice. An alternative is placing such selections after the hidden joins, as illustrated in Figure 5. This Post-filtering strategy is effective if the filtering can be done in a single pass over the result of the hidden joins. To meet this requirement, we use Bloom filters. The Bloom filter is a probabilistic bit array data structure that is used to test whether an element is a member of a set [3]. The two properties of Bloom filters are compactness and a very low false positive rate, making them well adapted to RAM-constrained environments.

Depending on the selectivities, a Pre-filtering or Post-filtering strategy can be selected per predicate. In addition, the selectivities of visible and hidden selections can be combined (Cross-filtering) before accessing a climbing index (resp. before

building a Bloom filter). Note also that the selectivity of a selection on intermediate tables of the join tree can be combined with the selectivity of selections on hidden attributes of descendant tables thanks to the climbing properties of the indexes. This leads to a large panel of candidate plans based on Pre-filtering, Post-filtering and Cross-Pre/Post-filtering.

5. DEMONSTRATION SCENARIO

The demonstration platform is an instance of the architecture presented in Figure 1. A first PC is used to host the public server and show traces of queries sent to it. A second PC hosts the client application, an applet downloaded from the USB device, and plays the role of the user’s terminal. This application can open connections with the public server and with the USB device. A third PC represents the USB device itself and serves as a secure display. The GhostDB prototype currently runs on a software simulator of the USB device and will be ported on the real hardware as soon as it is delivered to us by Gemalto, our industrial partner. Anyway, the demonstration GUI must run on a software simulator because the hardware device is by design unobservable.

We use a synthetic dataset compliant with the schema described in Figure 3. The cardinality of the root table (Prescription) is one million tuples.

A demo visitor interacts with the demo platform in three phases:

1. *Checking security*: see what is transferred among the three components of the platform during query execution.
2. *Testing query engine*: evaluate alternative query execution plans and operators internals.
3. *... and playing a game*: find the fastest plan for a query, with a prize to the best optimizer.

The target of the first phase is to present the rationale of the approach and show the benefits of GhostDB in terms of privacy

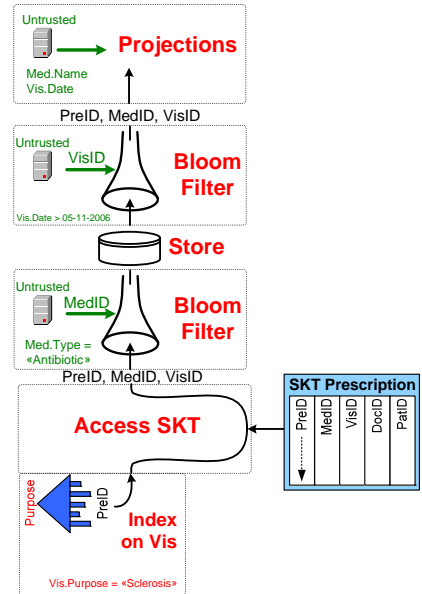


Figure 5. Post-filtering query execution plan.

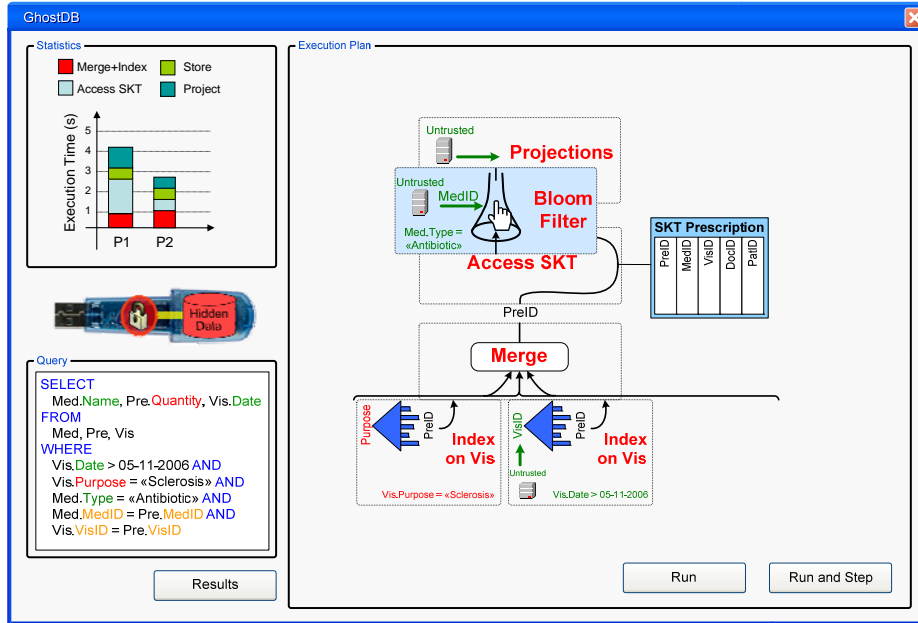


Figure 6. Building and evaluating ad-hoc query plans.

protection. First, we visualize the hidden and visible part of the database. Then, while running a query, the interface reveals what a pirate (e.g., Trojan horse) would observe, snooping the data transferred between the components of the architecture.

The second phase of the demo focuses on the novel query execution strategies tackling both the hardware constraints of the USB device and the combination of predicates on visible and hidden attributes. For a given query, the GUI allows the comparison of the relative performance of Pre-filtering and Post-filtering strategies in terms of RAM consumption and processing time (left upper side of Figure 6). To this end, the GUI helps building ad-hoc query execution plans, by easily modifying the ordering of high-level operators. A click on any plan operator (central part of Figure 6) displays a popup with additional statistics about this operator (number of processed tuples, local RAM consumption and processing time).

The last phase of the demo invites the visitors to assess their ability to select the best plan for a simple query. The rather unusual query execution strategies implemented in GhostDB may generate unexpected results for newcomers.

6. DEMO CONTRIBUTIONS

This demo makes three contributions: (i) it shows an architecture to achieve strong guarantees of data privacy using a small database on a RAM-limited but tamper-resistant USB device; (ii) it shows how to make this architecture achieve good performance through novel data structures and query processing strategies; and (iii) it allows the user to modify the query processing strategies to get insight into database computation on USB devices. Because the flash memory environment is increasingly popular, we think our techniques and the demo will be of general interest.

7. REFERENCES

- [1] Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P., Shasha, D., GhostDB: Querying Visible and Hidden data without leaks. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2007.
- [2] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y., Hippocratic Databases. *Int. Conf. on Very Large Databases (VLDB)*, 2002.
- [3] Bloom, B., Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13, 7, (1970).
- [4] Computer Security Institute., *CSI/FBI Computer Crime and Security Survey*. <http://www.gocsi.com>, 2006.
- [5] Desai, S., Netravali, A., Thompson, M., Carbon fibers as a novel material for high-performance microelectromechanical systems (MEMS). *Journal of Micromechanics and Microengineering*, 16, 7, (2006).
- [6] Hacigumus, H., Iyer, B., Li, C., Mehrotra, S., Executing SQL over Encrypted Data in the Database-Service-Provider Model. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2002.
- [7] Oracle Corporation. Oracle Database, Advanced Security Administrator's Guide, 10g Release 2 (10.2). Oracle documentation B14268-02, 2005.
- [8] Pucheral, P., Bouganim, L., Valduriez, P., Bobineau, C., PicoDBMS: Scaling down Database Techniques for the Smart Card. *Very Large Data Bases Journal*, 10, 3, (2001).
- [9] Sweeney, L., k-anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10, 5, (2002).
- [10] Vingralek, R., Gnatdb: A small-footprint, secure database system, *Int. Conf. on Very Large Databases (VLDB)*, 2002.
- [11] Weininger, A. Efficient execution of joins in a star schema. *ACM int. conf. on Management of data (SIGMOD)*, 2002.